SDF Workshops

Functors, Applicatives and Monads

Typeclasses

Many important types share important properties and methods such as `map` and `flatMap`.

Such shared methods and properties can be represented as type classes (classes of types).

Functors, Applicatives and Monads

`Functor`, `Applicative` and `Monad` are important classes that apply to types of **kind** `* => *`, such as `List`, `Option` and `Future`.

The classes are progressively more narrow:

```
Functor >: Applicative >: Monad
```

The types they contain are progressively more powerful!

Functor

`Functor` has a `map` method with this signature:

```
(A => B) => F[A] => F[B]
```

```
trait Functor[F[_]] {
  def map[A, B](f: A => B)(fa: F[A]) : F[B]
}
```

Roughly speaking, if a type `F` is a functor, you can apply functions **"inside"** `F`.

Functor laws

Also `map` must obey these *laws* (have these properties):

```
map(identity) === identity
```

```
map(f) compose map(g) === map (f compose g)
```

These allow us to safely refactor between these forms.

List is a Functor

```
implicit val listFunctor = new Functor[List] {
  def map[A, B](f: A => B)(fa: List[A]) : List[B] = fa.map(f)
}
```

Examples of the functor laws for `List` would then be:

```
List(1,2,3).map(x => x) === List(1,2,3)

List(1,2,3).map(_ * 2).map(_ + 1) === List(1,2,3).map(_ * 2 + 1)
```

More Functors

`Future, Stream, Option, Deserializer, Parser`

Types whose parameter can be covariant can be turned into functors (applications of `f` replace the upcast).

Types whose parameter can be contravariant cannot (Serializer, Ordering), but they would be cofunctors.

Applicative

`Applicative` has `apply` and `pure` methods with this signature:

```
F[A => B] => F[A] => F[B]
A => F[A]
```

```
trait Applicative[F[_]] {
  def apply[A, B](f: F[A => B])(fa: F[A]) : F[B]
  def pure[A](a: A) : F[A]
}
```

Roughly speaking, if a type `F` is an applicative, you can combine multiple instances of `F`, merging the values inside each in any way you like.

Applicative Extends Functor

```
F[A => B] => F[A] => F[B]
A => F[A]
// apply and pure are more powerful than map:
(A => B) => F[A] => F[B]

trait Applicative[F[_]] extends Functor[F] {
  def apply[A, B](f: F[A => B])(fa: F[A]) : F[B]
  def pure[A](a: A) : F[A]

  def map[A, B](f: A => B)(fa: F[A]) : F[B] = apply(pure(f))(fa)
}
```

Applicative Laws

- identity
- composition
- homomorphism
- interchange

List is an Applicative

```
implicit val listApplicative = new Applicative[List] {
  def apply[A, B](f: List[A => B])(fa: List[A]) : List[B] =
    fa.flatMap(a => f.map(_(a)))

  def pure[A](a: A) : List[A] = List(a)
}
```

The 4 applicative laws also hold.

More Applicatives

`Future, Stream, Option, Deserializer, Parser`

Not an applicative, only a functor:

```
case class Named[+A](name: String, a: A)
```

Monad

Monad has `flatMap` and `pure` methods with this signature:

```
(A => F[B]) => F[A] => F[B]
A => F[A]
```

```
trait Monad[F[_]] {
  def flatMap[A, B](f: A => F[B])(fa: F[A]) : F[B]
  def pure[A](a: A) : F[A]
}
```

Roughly speaking, if a type `F` is a monad, you can combine multiple instances of `F` , where some are chosen based on the values located inside the others.

Monad Extends Applicative

```
(A => F[B]) => F[A] => F[B]
A => F[A]
// flatMap and pure are more powerful than apply and pure:
F[A => B] => F[A] => F[B]

trait Monad[F[_]] extends Applicative[F] {
  def flatMap[A, B](f: A => F[B])(fa: F[A]) : F[B]
  def pure[A](a: A) : F[A]

  def apply[A, B](fab: F[A => B])(fa: F[A]) : F[B] =
    flatMap((a : A) => flatMap((ab : A => B) => pure(ab(a)))(fab))(fa)
}
```

Monad Laws

```
flatMap(f)(pure(a)) === f(a)

flatMap(pure) === identity

flatMap(g) compose flatMap(f) === flatMap(flatMap(g) compose f)
```

List is a Monad

```
implicit val listMonad = new Monad[List] {
  def flatMap[A, B](f: A => List[B])(fa: List[A]) : List[B] =
    fa.flatMap(f)

  def pure[A](a: A) : List[A] = List(a)
}
```

The 3 monad laws also hold.

More Monads

`Future`, `Stream`, `Option`, `Deserializer`, `Parser`

Many combinations of the above are also monads, such as

```
type FutureOption[A] = Future[Option[A]]
```

Not a monad, only an applicative: `FullTree`.

```
sealed trait FullTree[+A]
case class Leaf[+A](a: A) extends FullTree[A]
case class Node[+A](n: FullTree[(A, A)]) extends FullTree[A]
```